CS 4414 Assignment 4—Creating a User-Level Thread Package Part II

Due: 10pm Thursday, Mar. 28th

1. Project Description

In this project, you will gain experience with pre-emptive and priority scheduling, signals, and implementation of synchronization primitives. You can build on your own Part I or use the reference solution we provide.

2. Library Interface

Your assignment will be implemented in Linux using C. Recall that the old API calls to support were as follows; now the priority field in uthread_create will need to be supported.

- `void uthread_init()`

Initialize the thread library.

- `int uthread_create(void (*func)(int), int val, int pri)`

Create a new thread with priority `pri` start function `func` whose argument is `val`, i.e., the prototype of `func` is: `void func(int val)`. The `uthread_create` function returns the identifier of the thread thus created. **Now the priority field will need to be supported.** The orginal or "main" thread context should be given the lowest priority.

- `void uthread_yield()`

For the currently running thread to relinquish the CPU. See section 3 below for details on the new, pre-emptive, priority-aware scheduling policy.

- `void uthread_exit()`

Terminate the currently running thread and run the next thread, if there is one, according to the new, priority-aware scheduling policy. If the exiting thread is the only thread in the process, the entire process should exit. All threads, including the "main" (initial) context, should exit via uthread_exit(); behavior is undefined if any thread fails to do so (i.e., our test cases will always call uthread_exit()).

The new calls to support will be:

- `void uthread_mutex_init(uthread_mutex_t *lockVar)`

Initialize the previously allocated uthread_mutex_t `lockVar` — note that allocating memory for lockVar is the user's responsibility.

- `void uthread_mutex_lock(uthread_mutex_t *lockVar)`

The mutex object referenced by `lockVar` will be locked.[1] If the mutex is already locked, the calling thread **is suspended** until the mutex is available—**no busy-waiting allowed**!

- `void uthread_mutex_unlock(uthread_mutex_t *lockVar)`

Release the mutex object referenced by `lockVar` if no thread is waiting on it; otherwise, leave the mutex object locked but allow the highest-priority waiting thread to proceed.

- `void uthread_mutex_destroy(uthread_mutex_t *lockVar)`

Delete any dynamically allocated internal structures of `lockVar`. Do not free the uthread_mutex_t `lockVar` object itself; that is the user's responsibility.

[1] When we refer to a "mutex object," we mean a synchronization object that only has two states visible to the user: locked or free, and a queue of processes waiting to acquire the object.

3. Thread Scheduler

Every thread is given a time slice of 1 ms. When a time slice expires or a yield is called, a scheduling decision is made to find the next executable thread. Your library should support 10 different priority levels, with 0 being the highest and 9 the lowest. Use Round Robin scheduling to find the next runnable thread among threads of same priority level. Lower priority levels should not be serviced unless there are no runnable threads at higher levels. For simplicity, **priorities are fixed**, i.e., feedback scheduling is **not allowed**.

4. Useful UNIX System Calls

• **For thread scheduling:** Use `getitimer()` and `setitimer()` with ITIMER_VIRTUAL to catch `SIGVTALRM`. The signal handler would implement the necessary thread scheduling functionality. Do not use signal() to register a signal handler; use sigaction() instead. You may use either the one-argument or three-argument signal handler. If you use the ucontext provided by the three-argument handler, you must manually set its stack pointer and stack size fields before using the context; the signal facility does not set these.

• **Mutex lock:** The lock and unlock operations have to be *atomic*. Atomic execution can be enforced by disabling interrupts using `sigprocmask()/sigaction()` or related functions.

You may not use primitives from the OS that implement the above functionalities, or existing thread packages (eg semaphores), other than signal/timer functions. You also may not implement lock/unlock by simply blocking signals for the duration between lock and unlock (although you probably want to block them at the beginning of each of those functions and unblock them before returning). That would defeat the purpose of this assignment. You must implement the scheduler and mutual-exclusion primitives yourself. Please ask if you have questions!

5. What to Turn in

Remember: we'd rather see something basic that works rather than something sophisticated that doesn't work at all. Get the most rudimentary functionality working first and build from there.

a. A report (at most 5 pages) with a detailed description of how each of the functionalities above for the uthread package was implemented, the design decisions you made, and an explanation of why you made those decisions.

b. Please upload a single gzipped tar file on Collab containing your report, code, and your PARTNERS file, which includes the pledge and both partners' name. You should also include a separate text file named CONTRIBUTIONS that briefly describes the contributions of each team member.

The code should build "threadpkg2.o" from a Makefile at the command line in the VM environment when we enter "make threadpkg2".

Assignment adapted from Prof. Sudhanva Gurumurthi